

信息论与字谜游戏-信息在决策中的关键作用

翁正朗 PB22000246 牛博轩 PB22061271 刘沛 PB22061259 孔令茹 PB22231827

中国科学技术大学

摘要 : 本文从信息熵和KL散度的角度研究了信息在决策过程中发挥的关键作用

关键词 : 熵, KL散度, 字谜, 信息

1 引言

1.1 问题背景

1.2 任务称述

1.2.1 游戏规则

1.2.2 单词模式匹配机制介绍

1.3 难度的升级

2 基本任务解决方案

2.1 信息的衡量

2.2 算法

2.3 实验优化

2.4 代码:

3 实验与实验结果

4 扩展与发散

4.1 任务描述

4.1.1 游戏规则

4.1.2 模式匹配范式

4.1.3 目标:

4.2 任务解决方案

4.2.1 匹配模式分布的数学建模

4.2.2 实验上的假设

4.2.3 实验优化

4.3 实验结果

4.4 KL散度的数学解释

5 附录

5.1 A 基本问题代码

5.2 B 发散问题代码

1 引言

1.1 问题背景

Wordle 目前是《纽约时报》提供的一款流行的每日谜题，每一天游戏会选取一个固定的单词作为谜底，已有 60 多个版本。玩家可以选择“普通模式”或“困难模式”。玩家尝试在 6 次或更少的尝试中猜出一个 5 字母单词，每次猜测会收到反馈，方块的颜色会发生变化（绿色、黄色、灰色）。注意：每次猜测必须是英语中的真实单词，更具体地说，是游戏认可的合法输入中的单词。

绿色方块表示该方块中的字母在单词中且位置正确。黄色方块表示该方块中的字母在单词中但位置错误。灰色方块表示该方块中的字母不在单词中。

我们将从信息论的角度，利用信息熵和KL散度的知识，编写代码求解该问题及其扩展情况。借此契机，进一步思考信息论在决策中的应用。

1.2 任务称述

可作为合法输入的单词全集（可用集） \mathcal{S} ，大小为 N ，其中，每个单词的长度都是5。可作为答案的单词集合（答案集） \mathcal{A} ：

$$\mathcal{A} \subseteq \mathcal{S}, |\mathcal{A}| = n, |\mathcal{S}| = N$$

1.2.1 游戏规则

玩家每次开始进行游戏，系统将从答案集当中随机（等概率）抽取一个答案，并在接下来的连续试错过程中保持不变，直到玩家猜对或者放弃且开始下一个单词的猜测。

我们将这种模式下的游戏描述为：

一轮游戏：以系统从答案集中抽取一个单词作为谜底开始，以玩家猜对或者选择放弃猜测作为结束。

第 i 轮的谜底记为 $\mathbf{a}^i = (a_1, \dots, a_5), a_i \in \text{alphabet}$

一次猜测：在第 i 轮游戏的第 k 次猜测中，玩家选定一个单词，作为玩家对谜底的猜测，并提交到系统中，系统将给出相应的真实谜底 \mathbf{a}^i 与猜测的单词 $\hat{\mathbf{a}}_k^i$ 匹配结果 $\mathbf{r} = f(\mathbf{a}^i, \hat{\mathbf{a}}_k^i) \in R, |R| = 3^5$

其中 R 表示所有反馈模式。注意区分一轮游戏（以上标表示），和一次猜测（用下标表示）。

1.2.2 单词模式匹配机制介绍

系统在玩家提交每一次猜测后，都会给出真实谜底 \mathbf{a}^i 与猜测的单词 $\hat{\mathbf{a}}_k^i$ 匹配结果

$$\mathbf{r} = f(\mathbf{a}^i, \hat{\mathbf{a}}_k^i) = (r_1, r_2, \dots, r_5), r_j \in \{0, 1, 2\}$$

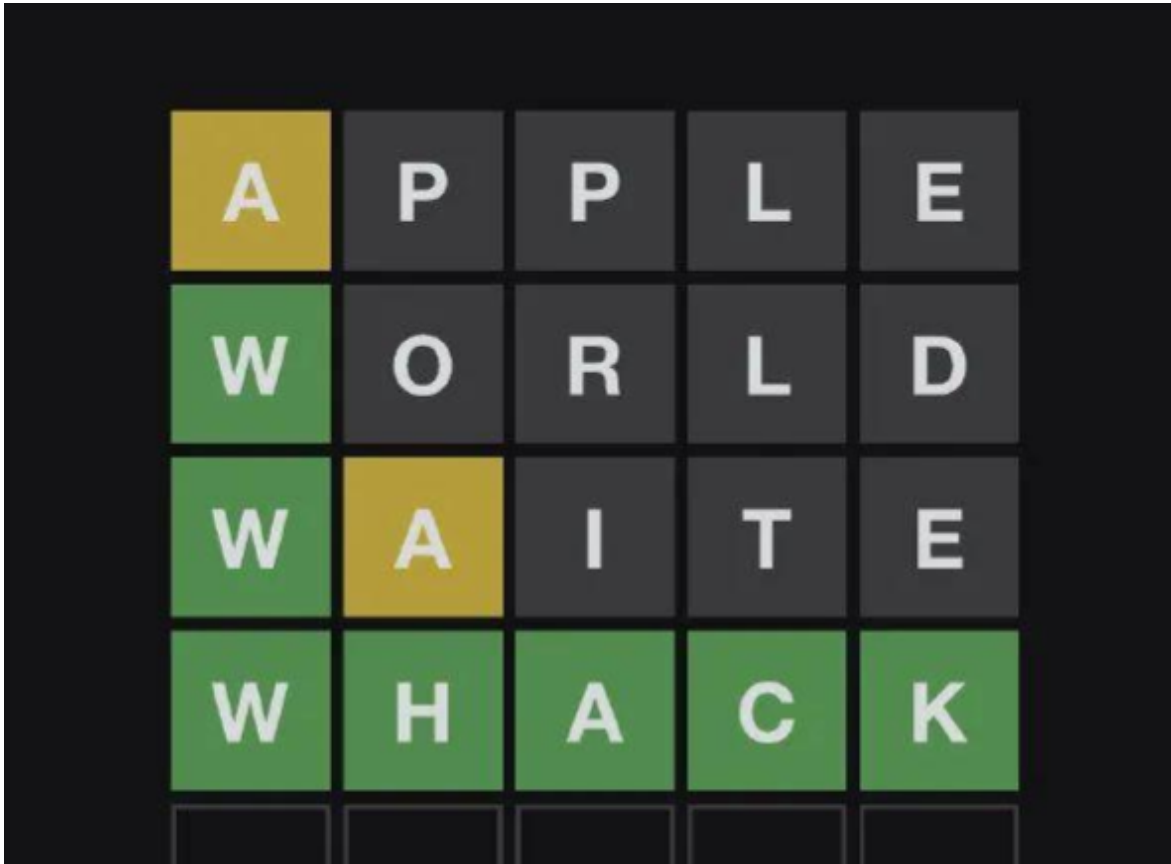
其中， r_j 表示两个单词位置 j 处的字母相关信息：

0:灰色，表示玩家提交的猜测单词 j 处的字母与真实谜底 j 处的字母不相同，且猜测单词 j 处的字母在真实谜底当中未出现

1:黄色，表示玩家提交的猜测单词 j 处的字母与真实谜底 j 处的字母不相同，但猜测单词 j 处的字母在真实谜底当中出现过，重复出现的，只算做出现一次

2:绿色，表示玩家提交的猜测单词 j 处的字母与真实谜底 j 处的字母相同

示例



简单的颜色匹配情况不会有争议。



上图为容易混淆的重复字母匹配情况，注意区分预测答案中重复出现的字母是否在真实谜底中也重复出现，真实谜底中对于字母数量（无论是绿色还是黄色），都要求严格匹配。

1.3 难度的升级

1. 只给出该位置的字母正确或错误的提示（信息量的减少）
2. 不给出答案集（信息量的减少）
3. 更一般的情况，每个单词作为答案的概率不同（信息量的增加）

2 基本任务解决方案

基本思路：每次猜词我们都将得到部分信息，基于贪心的想法，我们每次猜词选择能获得的“信息”最多的词。

2.1 信息的衡量

对于每个答案 \mathbf{a} ，输入一个可行猜测 $\mathbf{s} \in S$ 会得到一个反馈模式，即得到一定的信息。如果这个反馈模式在对于答案 \mathbf{a} 可得到的所有反馈模式中出现概率很小，则选取得到这个反馈模式的猜测会给我们提供最多的信息。（考虑猜测的单词就是答案，则得到反馈模式 $[2, 2, 2, 2, 2]$ ，该模式出现频数为1，给我们提供了最多的信息：明确告知了答案是什么）

2.2 算法

每次猜测后，合法输入集（可用集） S 是不变的。记第 i 次猜测结束后，剩余可行的答案集为 A_i ， $A_0 = A$ ， $i \in \mathbb{N}$ ，则显然 $A = A_0 \supseteq A_1 \supseteq A_2 \supseteq \dots$

算法如下：

在第 $i+1$ 轮猜测时，计算 $\mathbf{s} \in S$ 作为猜测所获得的反馈 $\mathbf{r}_s \in R$

1. 对每个 $\mathbf{a} \in A_i$ ，假设以此单词作为答案
2. 遍历 $\forall \mathbf{a} \in A_i$ ，计算以 \mathbf{a} 为答案， \mathbf{s} 为猜测时的反馈模式 \mathbf{r}_a
3. 统计每个 \mathbf{r}_a 的出现频率，以频率代替概率，计算反馈模式的分布的熵

$$H_{s, A_i}(\mathbf{r}) = - \sum_{\mathbf{r} \in R} \frac{N(\mathbf{r}|\mathbf{s}, A_i)}{|A_i|} \log \frac{N(\mathbf{r}|\mathbf{s}, A_i)}{|A_i|}$$

其中 $N(\mathbf{r}|\mathbf{s}, A_i)$ 表示以 \mathbf{s} 为猜测时，剩余可能答案 A_i 得到的所有反馈模式中，反馈模式 \mathbf{r} 的频数。

遍历 $\forall \mathbf{s} \in S$ ，选取

$$\hat{\mathbf{s}}_{i+1}^* = \arg \max_{\mathbf{s} \in S} H_{\mathbf{s}, A_i}(\mathbf{r})$$

作为第 $i+1$ 轮的猜测输入。

2.3 实验优化

上述算法中直接以频率代替概率，即假设了以等概率从 A 中选取答案单词。若可以得到选词的真实概率，如自然语言中的词频，或wordle游戏以往的答案选词频率。则可给 $N(\mathbf{r}|\mathbf{s}, A_i)$ 乘以此频率权重，重新计算每次的熵。
















2.4 代码：

[请参考附录A](#)

3 实验与实验结果

```
1 | input a target word: abyss
2 | Loading precomputed feedback patterns...
3 | Total input words: 12953
4 | Total answer words: 2309
5 | Target word: abyss
6 | Top 10 words by entropy:
7 | raise: 5.8783
```

```

8      slate: 5.8558
9      crate: 5.8352
10     irate: 5.8328
11     trace: 5.8304
12     arise: 5.8210
13     stare: 5.8069
14     snare: 5.7687
15     arose: 5.7678
16     least: 5.7516
17     Enter your guess: slate
18     Guess 1: slate     
19     Possible words remaining: 14
20     Top 10 words by entropy:
21     amiss: 3.5216
22     abyss: 3.3788
23     basin: 3.3788
24     pansy: 3.2359
25     arson: 3.1820
26     assay: 3.1820
27     daisy: 3.1820
28     gassy: 3.1820
29     basis: 3.1820
30     marsh: 3.1281
31     Enter your guess: amiss
32     Guess 2: amiss     
33     Possible words remaining: 1
34     Top 10 words by entropy:
35     abyss: 0.0000
36     Enter your guess: abyss
37     Guess 3: abyss     
38     Congratulations! You've guessed the word!

```

计算不同初始单词策略下的平均猜测次数:

```
1 Loading precomputed feedback patterns...
2 First guess: salet
3 Processing words: 100%|██████████████████████████████████████| 2309/2309 [00:16<00:00,
  141.67it/s]
4 Average attempts to solve: 3.53
```

4 扩展与发散

如果我们依旧考虑wordle的大背景，但是调整游戏规则如下：

4.1 任务描述

已知的单词全集（可用集） \mathcal{S} ,大小为N，其中，每个单词的长度都是5

未知的答案集 \mathcal{A} :

$$\mathcal{A} \subseteq \mathcal{S}, |\mathcal{A}| = n$$

4.1.1 游戏规则

字谜游戏当中，每一次出题，系统都会随机（等概率）从答案集中抽取一个单词作为谜底；**谜底每次都会变！**，所以玩家只有一次猜测谜底的机会。

玩家可以从全集当中挑选一个单词作为玩家对谜底的猜测提交给系统，系统将给出玩家猜测和真实答案的匹配向量 \mathbf{r}

4.1.2 模式匹配范式

$$\text{result} = f(\text{word}_1, \text{word}_2) = \mathbf{r} = (r_1, r_2, \dots, r_5), \quad r_i \in \{0, 1, 2\}$$

模式匹配的机制与原始问题中的相同。

[查看猜词语模式匹配机制](#)

4.1.3 目标：

通过最少数量的尝试，将整个答案集拟合出来，将单次猜词正确率尽可能提高。

4.2 任务解决方案

1. 从全集中随机挑选 n （也就是答案集的大小）个单词，充当我们对答案集合的估计 $\hat{\mathcal{A}}_0 = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_n\}$
2. 从全集中再随机挑选 c 个单词，组成测试集合 \mathcal{S} 。每个单词都将作为字谜游戏的输入，同时系统会给出相应的模式匹配向量 \mathbf{r} ；我们规定这 c 个单词中的每一个，都输入 M 次；根据这 $M \times c$ 个模式匹配向量，我们会得到一个**实验分布** $E_0(\text{word}_1, \dots, \text{word}_c)$

3. 同时，我们根据 $\hat{\mathcal{A}}_0$ 和 \mathcal{S} 得到**估计分布** $P_0(\text{word}_1, \dots, \text{word}_c)$ ，我们定义损失函数loss为：

$$L_0 = D(E_0 || P_0)$$

4. 计算：损失函数关于估计答案集中单词 \mathbf{a}_i 的离散梯度，

$$\Delta(\mathbf{a}_i) = L_0 - D(E_0 || P_0^{\setminus \mathbf{a}_i})$$

其中 $P_0^{\setminus \mathbf{a}_i}$ 表示，从估计答案集中剔除单词 \mathbf{a}_i 后，重新计算得到的测试集合在估计答案集合上的分布。

并找到 Δ 最大的 m 个估计答案，从 $\hat{\mathcal{A}}_0$ 中舍弃，再从全集中随机挑选 m 个单词，加入到 $\hat{\mathcal{A}}$ 中，得到 $\hat{\mathcal{A}}_1$ ；

5. 回到步骤2。开始迭代

$$\{\hat{\mathcal{A}}_k P_k, L_k\} \rightarrow \{\hat{\mathcal{A}}_{k+1}, P_{k+1}, L_{k+1}\}$$

直到LOSS够小

4.2.1 匹配模式分布的数学建模

其中，分布 $E(\text{word})$ ，会是一个向量；表示word输入到系统中 M 次， M 个模式匹配向量的分布，我们用经验分布来建模单词单词word的匹配模式分布。

$$E(\text{word})_{\mathcal{A}} = \{ \text{' } \blacksquare \blacksquare \blacksquare \blacksquare \blacksquare \text{' } : \frac{x_1}{M}, \text{' } \blacksquare \blacksquare \blacksquare \blacksquare \blacksquare \text{' } : \frac{x_2}{M}, \dots, \text{' } \blacksquare \blacksquare \blacksquare \blacksquare \blacksquare \text{' } : \frac{x_j}{M}, \dots, \text{' } \blacksquare \blacksquare \blacksquare \blacksquare \blacksquare \text{' } : \frac{x_{53}}{M} \}$$

其中， x_j 代表第 j 种模式出现的次数， M 是以单词word作为谜底进行实验的次数

$$\text{同理，} P_i(\text{word})_{\hat{\mathcal{A}}} = \{ \text{' } \blacksquare \blacksquare \blacksquare \blacksquare \blacksquare \text{' } : \frac{x_1}{n}, \text{' } \blacksquare \blacksquare \blacksquare \blacksquare \blacksquare \text{' } : \frac{x_2}{n}, \dots, \text{' } \blacksquare \blacksquare \blacksquare \blacksquare \blacksquare \text{' } : \frac{x_j}{n}, \dots, \text{' } \blacksquare \blacksquare \blacksquare \blacksquare \blacksquare \text{' } : \frac{x_{53}}{n} \}$$

其中， x_j 代表第 j 种模式出现的次数， n 是以单词word在估计答案集中做匹配的次数，因为估计答案集合大小为 n ，所以需要 n 次匹配。

4.2.2 实验上的假设

我们在计算实验分布的时候，实际上是期望得到真实答案集的分布，在实验时不妨直接在答案集上采样(也就是做 $c \times n$ 次模式匹配)，便于我们进行叙述和实验进行。

如果用采样实验采样代替真实分布，效果会出现一定程度的下降，但是随着采样次数够大，采样分布也会趋近于真实分布。

4.2.3 实验优化

考虑到答案集在全集的占比会比较小，每次随机从全集里选择 m 个单词进行估计答案集的更新，收敛速度较慢，我们会在第一步进行全集的一个简单过滤。

当我们知道 $E(\text{word}_1, \dots, \text{word}_c)$ 之后，我们将测试集在全集上做一次模式匹配。我们将得到全集中任意一个单词 word 在测试集上的模式分布：

$$\mathcal{T}_i(\text{word})_{\mathcal{S}} = \{ \text{模式1} : \frac{x_1}{c}, \text{模式2} : \frac{x_2}{c}, \dots, \text{模式j} : \frac{x_j}{c}, \dots, \text{模式53} : \frac{x_{53}}{c} \}$$

我们只要发现单词 word 在测试集上的任意一个模式匹配结果没有出现在测试集在与真实答案做匹配得到的 $c \times n$ 个模式当中，那么可以确定 word 不可能出现在答案集中。

简单证明

我们采用反证的方法：

假设 word 是答案集的一员，那么 $r(\text{word}, \text{test})$ 必然和测试集 ($\text{test} \in \mathcal{S}$) 在与真实答案 ($\text{word} \in \mathcal{A}$) 做匹配得到的 $c \times n$ 个模式中的一个相同，与前提矛盾，可得“ word 是答案集的一员”这个假设不对

所以 word 不可能在答案集中。

4.3 实验结果

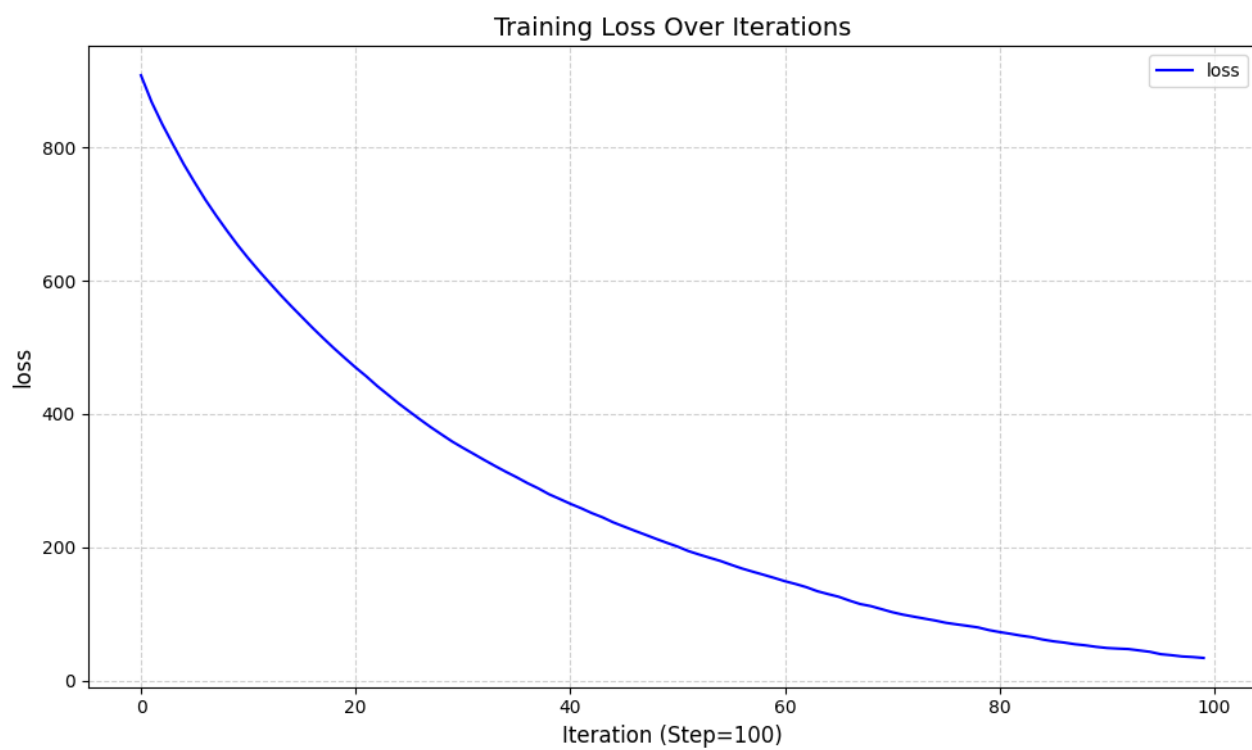
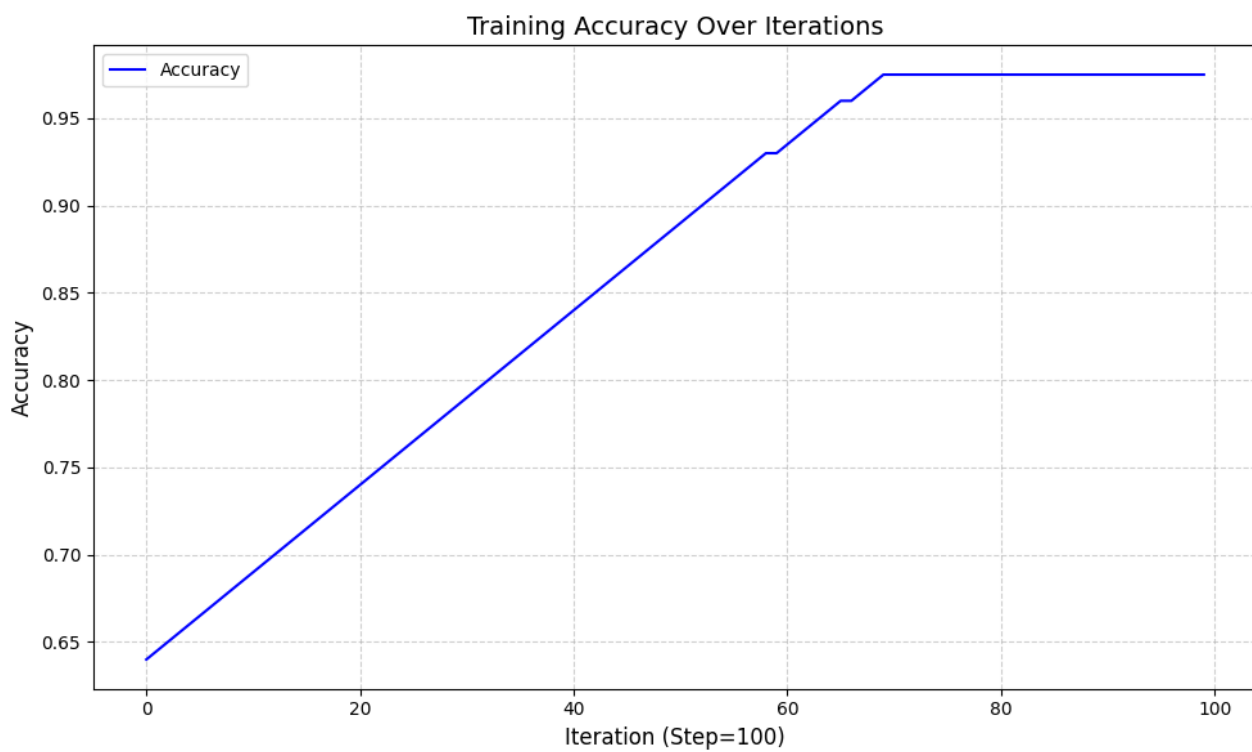
我们的全集一共有2309个五个字母的单词，我们从中随机抽取200个单词作为答案集。

1. 训练过程曲线

测试集从全集随机抽取30个，每次迭代的时候，从估计答案集合中替换更新1个单词。

在进行迭代更新之前，我们通过预先设计的方法过滤全集得到的候选集大小为436，多次实验表明，这个数字在400上下浮动。

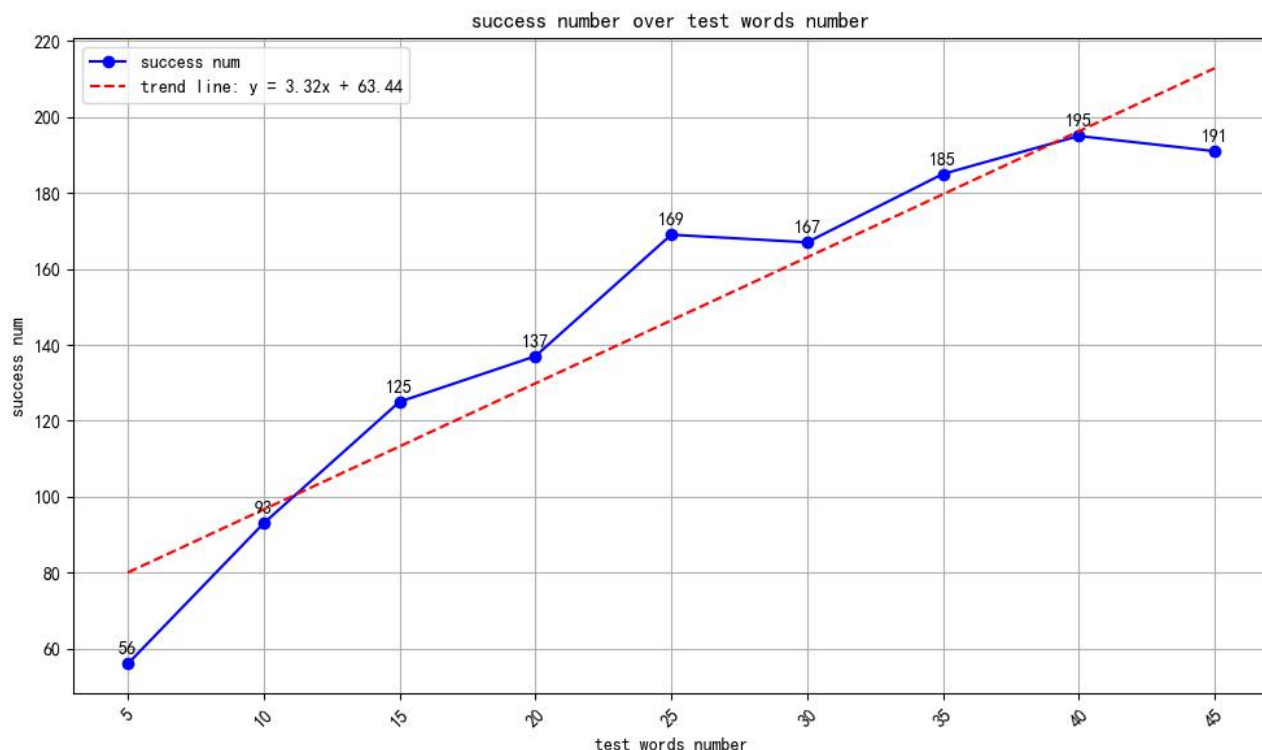
迭代次数100次：



可以见得，这个利用KL散度进行梯度下降迭代的方法，效果十分不错。

2.更改测试集大小，对于实验结果的影响

用来作实验的测试集的大小代表我们对于答案集的了解程度，实验集越大，掌握的信息就越多，理论上剩余不确定的熵就越小。



可以看到，实验和理论预期一致，随着用来做实验单词个数增加，我们对于分布的掌握更加精确，得到答案单词的概率也就越高。

4.4 KL散度的数学解释

这里，我们用泛函的视角重新理解信息论中各种熵的定义。

可以粗浅地将泛函理解为函数的函数，如： $[a, b]$ 区间上函数 $f(x)$ 的定积分 $L: f \in \mathcal{F} \mapsto \mathbb{R}$ 就是函数 $f(x)$ 的一种泛函。 L 将函数空间 \mathcal{F} 中的函数 f 映射到实数域 \mathbb{R} 。

设 \mathcal{P} 表示所有分布的集合， $p \in \mathcal{P}$ 为一个分布，则实际上，信息熵为分布函数的泛函：

$$H(p) = \sum_x -p(x) \log_2 p(x) = \int_x -p(x) \log p(x) dx$$

$H(X)$ 与 \mathbf{r}, \mathbf{v} 、 X 的具体取值无关，只与其分布函数 p 有关。也就是说，如上定义了一个泛函 $H: \mathcal{P} \mapsto \mathbb{R}$

类似地，KL散度（相对熵）也是一种泛函，只不过定义在两个分布之上： $D: \mathcal{P} \times \mathcal{P} \mapsto \mathbb{R}$ 。对于 $p, q \in \mathcal{P}$:

$$D(p||q) = \sum_x p(x) \log \frac{p(x)}{q(x)}$$

相对熵定义了一种非传统意义下的“距离”，于是，在这种“距离”下，就可以进行泛函最优化问题。如前文提到的

$$\min L_0 = D(E_0||P_0)$$

Thm.11.4.1 (Sanov) X_1, \dots, X_n i.i.d. $\sim Q(x) \in \mathcal{P}, E \subseteq \mathcal{P}$, then

$$\begin{aligned} Q(E) &= \sum_{\mathbf{x}: P_{\mathbf{x}} \in E \cap \mathcal{P}_n} Q(\mathbf{x}) \\ &\leq (n+1)^{|\mathcal{X}|} 2^{-nD(p^*||q)} \end{aligned}$$

where

$$p^* = \arg \min_{p \in E} D(p||q)$$

Additionally, if $E = \text{cls int } E$, then

$$\lim_{n \rightarrow \infty} \frac{1}{n} \log Q(E) = -D(p^*||q)$$

再考虑上面提到的，每次随机抽 c 个单词得到的实验分布 E_0 ，和用估计的答案集得到的估计分布 P_0 ，优化问题即为

$$P^* = \arg \min_{P_0 \in E} D(P_0||E_0)$$

也就是想在 E 中找一个分布函数 P_0 ，最小化 $D(P_0||E_0)$ ，这本质上是一个泛函最优化问题。

5 附录

5.1 A 基本问题代码

```
1 import random
2 import math
3 from collections import Counter
4 import numpy as np
5 from tqdm import tqdm
6
7 def load_words(filename):
8     with open(filename, 'r') as file:
9         return [line.strip() for line in file] #•文件处理•: 读取文件时去除每行末尾的换行
10 符:
11
12 # input_words_path = 'valid-words14855.txt'
13 # answer_words_path = 'answer_words.txt'
14
15 input_words_path = './data/allowed_words.txt'
16 answer_words_path = './data/possible_words.txt'
17 feedback_patterns_path = './feedback_patterns.npy'
18
19 # 加载输入单词和答案单词
20 input_words = load_words(input_words_path)
21 answer_words = load_words(answer_words_path)
22
23 def GetTargetWord():
24     while 1:
25         target_word = input("input a target word: ").strip().lower()
26         if len(target_word) != 5:
27             print("Please enter a 5-letter word.")
28             continue
29         if target_word not in answer_words:
30             print(f"Warning: Target word '{target_word}' is not in the answer words
31 list.")
32             continue
33         break
34     return target_word
35
36 def display_feedback(guess, target): #这个检验是对的
```

```

35     feedback = ["□"] * 5
36     target_letter_count = {}    #empty dict
37
38     # First pass: check for correct positions
39     for i in range(5):
40         if guess[i] == target[i]:
41             feedback[i] = "■"
42         else:
43             if target[i] in target_letter_count:
44                 target_letter_count[target[i]] += 1
45             else:
46                 target_letter_count[target[i]] = 1
47
48     # Second pass: check for correct letters in the wrong positions
49     for i in range(5):
50         if feedback[i] == "□" and guess[i] in target_letter_count and
target_letter_count[guess[i]] > 0:
51             feedback[i] = "■"
52             target_letter_count[guess[i]] -= 1
53
54     return ''.join(feedback)
55
56 def display_feedback_encode(guess, target):
57     feedback = [0] * 5    # 0: gray, 1: yellow, 2: green
58     target_letter_count = {}
59
60     # First pass: check for correct positions
61     for i in range(5):
62         if guess[i] == target[i]:
63             feedback[i] = 2    # Green
64         else:
65             if target[i] in target_letter_count:
66                 target_letter_count[target[i]] += 1
67             else:
68                 target_letter_count[target[i]] = 1
69
70     # Second pass: check for correct letters in the wrong positions
71     for i in range(5):
72         if feedback[i] == 0 and guess[i] in target_letter_count and
target_letter_count[guess[i]] > 0:
73             feedback[i] = 1    # Yellow
74             target_letter_count[guess[i]] -= 1
75
76     return feedback
77
78 def filter_words(feedback_patterns, words, guess, feedback):
79     possible_words = []
80     for word in words:
81         match = True
82
83         test_feedback = feedback_patterns[word][guess]
84         if test_feedback != feedback:
85             match = False
86             continue
87
88         if match:

```

```

89         possible_words.append(word)
90
91     return possible_words
92
93
94 def calculate_entropy(possible_words, guess):
95     feedback_counts = Counter(display_feedback(guess, word) for word in possible_words)
96     total = sum(feedback_counts.values())
97     entropy = 0.0
98     for count in feedback_counts.values():
99         probability = count / total
100         entropy -= probability * math.log2(probability)
101     return entropy
102
103 # 修改熵计算函数，使用预计算的反馈模式
104 def calculate_entropy_with_precomputed(feedback_patterns, possible_words, guess):
105     feedback_counts = Counter(
106         tuple(feedback_patterns[target][guess]) for target in possible_words
107     )
108     total = sum(feedback_counts.values())
109     entropy = 0.0
110     for count in feedback_counts.values():
111         probability = count / total
112         entropy -= probability * math.log2(probability)
113     return entropy
114
115 def display_top_entropy_words(possible_words):
116     entropies = [(word, calculate_entropy(possible_words, word)) for word in
possible_words]
117     entropies.sort(key=lambda x: x[1], reverse=True)
118
119     print("Top 10 words by entropy:")
120     for word, entropy in entropies[:10]:
121         print(f"{word}: {entropy:.4f}")
122
123 def wordle_game():
124     max_guesses = 6
125     guess_count = 0
126     possible_words = answer_words
127     target_word = GetTargetWord()
128
129     # 打印单词数和目标单词
130     print(f"Total input words: {len(input_words)}")
131     print(f"Total answer words: {len(answer_words)}")
132     print(f"Target word: {target_word}")
133
134     # 初次计算并显示熵分布
135     display_top_entropy_words(possible_words)
136
137     while guess_count < max_guesses:
138         guess = input("Enter your guess: ").strip().lower()
139
140         if len(guess) != 5:
141             print("Please enter a 5-letter word.")
142             continue
143

```

```

144         if guess not in input_words:
145             print("Word not in list.")
146             continue
147
148         guess_count += 1
149         feedback = display_feedback(guess, target_word)
150         print(f"Guess {guess_count}: {guess} {feedback}")
151
152         if guess == target_word:
153             print("Congratulations! You've guessed the word!")
154             return
155
156         possible_words = filter_words(possible_words, guess, feedback)
157         print(f"Possible words remaining: {len(possible_words)}")
158
159         display_top_entropy_words(possible_words)
160
161     print(f"Game Over. The word was: {target_word}")
162
163
164 def wordle_game_simulation():    #暂时废弃
165     total_rounds = 0
166     num_simulations = len(answer_words)
167
168     for target_word in answer_words:
169         guess_count = 0
170         possible_words = input_words
171
172         while True:
173             # 计算每个可能单词的熵
174             entropies = [(word, calculate_entropy(possible_words, word)) for word in
possible_words]
175             entropies.sort(key=lambda x: x[1], reverse=True)
176
177             # 选择熵最大的单词作为猜测
178             guess = entropies[0][0]
179             guess_count += 1
180
181             feedback = display_feedback(guess, target_word)
182
183             if guess == target_word:
184                 total_rounds += guess_count
185                 break
186
187             possible_words = filter_words(possible_words, guess, feedback)
188
189     average_rounds = total_rounds / num_simulations
190     print(f"Average rounds to solve: {average_rounds:.2f}")
191
192 def wordle_game_with_precomputed():
193     max_guesses = 6
194     guess_count = 0
195     possible_words = answer_words
196     target_word = GetTargetWord()
197
198     # 加载预计算的反馈模式

```

```

199     print("Loading precomputed feedback patterns...")
200     feedback_patterns = np.load(feedback_patterns_path, allow_pickle=True).item()
201
202     print(f"Total input words: {len(input_words)}")
203     print(f"Total answer words: {len(answer_words)}")
204     print(f"Target word: {target_word}")
205
206
207     # 初次计算并显示熵分布
208     # display_top_entropy_words(possible_words)
209     # 使用预计算数据计算熵
210     entropies = [
211         (word, calculate_entropy_with_precomputed(feedback_patterns, possible_words,
212 word))
213     ]
214     entropies.sort(key=lambda x: x[1], reverse=True)
215
216     print("Top 10 words by entropy:")
217     for word, entropy in entropies[:10]:
218         print(f"{word}: {entropy:.4f}")
219
220     while guess_count < max_guesses:
221         guess = input("Enter your guess: ").strip().lower()
222
223         if len(guess) != 5:
224             print("Please enter a 5-letter word.")
225             continue
226
227         if guess not in input_words:
228             print("Word not in list.")
229             continue
230
231         guess_count += 1
232         feedback = display_feedback(guess, target_word)
233         print(f"Guess {guess_count}: {guess} {feedback}")
234
235         if guess == target_word:
236             print("Congratulations! You've guessed the word!")
237             return
238
239         feedback_encode = display_feedback_encode(guess, target_word)
240         possible_words = filter_words(feedback_patterns, possible_words, guess,
241 feedback_encode)
242         print(f"Possible words remaining: {len(possible_words)}")
243
244         # 使用预计算数据计算熵
245         entropies = [
246             (word, calculate_entropy_with_precomputed(feedback_patterns, possible_words,
247 word))
248         ]
249         entropies.sort(key=lambda x: x[1], reverse=True)
250
251         print("Top 10 words by entropy:")
252         for word, entropy in entropies[:10]:

```

```

252         print(f"{word}: {entropy:.4f}")
253
254     print(f"Game Over. The word was: {target_word}")
255
256 def wordle_game_average_attempts(feedback_patterns, first_guess = 'crane'):
257     total_attempts = 0
258     num_words = len(answer_words)
259     print(f"First guess: {first_guess}")
260
261
262     for target_word in tqdm(answer_words, desc="Processing words"):
263         guess_count = 0
264         possible_words = answer_words
265
266         # 第一轮猜测固定为 "raise"
267         guess = first_guess
268         feedback = display_feedback_encode(guess, target_word)
269         guess_count += 1
270
271         if guess == target_word:
272             total_attempts += guess_count
273             continue
274
275         possible_words = filter_words(feedback_patterns, possible_words, guess, feedback)
276
277         while True:
278             # 使用预计算数据计算熵
279             entropies = [
280                 (word, calculate_entropy_with_precomputed(feedback_patterns,
possible_words, word))
281                 for word in possible_words
282             ]
283             entropies.sort(key=lambda x: x[1], reverse=True)
284
285             # 选择熵最大的单词作为猜测
286             guess = entropies[0][0]
287             feedback = display_feedback_encode(guess, target_word)
288             guess_count += 1
289
290             if guess == target_word:
291                 total_attempts += guess_count
292                 break
293
294             possible_words = filter_words(feedback_patterns, possible_words, guess,
feedback)
295
296     average_attempts = total_attempts / num_words
297     print(f"Average attempts to solve: {average_attempts:.2f}")
298
299 if __name__ == "__main__":
300     # 加载预计算的反馈模式
301     print("Loading precomputed feedback patterns...")
302     feedback_patterns = np.load(feedback_patterns_path, allow_pickle=True).item()
303     wordle_game_average_attempts(feedback_patterns, first_guess='salet')
304
305     # wordle_game_with_precomputed()

```

5.2 B 发散问题代码

```
1
2 import random
3 import math
4 from collections import Counter
5
6 from matplotlib import pyplot as plt
7
8 def load_words(filename):
9     with open(filename, 'r') as file:
10         return [line.strip() for line in file]
11
12 # 加载输入单词和答案单词
13 #input_words = load_words('valid-words14855.txt')
14 answer_words = load_words('answer_words.txt')
15 print(f"共{len(answer_words)}")
16 target_num=200
17 target_words=random.sample(answer_words,target_num)
18 print(target_words)
19 guess_words=random.sample(answer_words,30)
20 guess_time=5*10
21
22
23 def display_vector(guess, target):
24     """
25     guess : 输入的猜测
26     target:真实谜底
27
28     return: feedvack
29     """
30     feedback = ["0"] * 5
31     target_letter_count = {}
32
33     # First pass: check for correct positions
34     for i in range(5):
35         if guess[i] == target[i]:
36             feedback[i] = "2"
37         else:
38             if target[i] in target_letter_count:
39                 target_letter_count[target[i]] += 1
40             else:
41                 target_letter_count[target[i]] = 1
42
43     # Second pass: check for correct letters in the wrong positions
44     for i in range(5):
45         if feedback[i] == "0" and guess[i] in target_letter_count and
46 target_letter_count[guess[i]] > 0:
47             feedback[i] = "1"
48             target_letter_count[guess[i]] -= 1
49
49     return ''.join(feedback)
```



```

50
51 def caculate_answer_dict(guess_dict):
52     #这是,
53     answer_dict=[]
54     for ans in answer_words: #遍历全集
55         ansdict={'word':ans}
56         for guess in guess_dict:
57             word=guess['word']
58             vector=display_vector(word,ans) #给出ans与全部guess的匹配
59
60             if vector not in guess.keys():#既然guess里面包含了对整个答案集的匹配,那么如果这
一个单词ans, 只要
61                 #对于任意一个guess的全部匹配都不同, 那这个ans就不可能是答案集的一员
62                 ansdict=None
63                 break
64                 ansdict[word]=vector
65             if(ansdict!= None):
66                 answer_dict.append(ansdict)
67         print(len(answer_dict))
68         #这是排除了没可能出现在答案集当中的单词的全集, 新的浓缩全集
69         # for ans in answer_dict:
70         #     print(ans['word'])
71         return answer_dict
72
73 def caculate_kl(test_dict,guess_dict):
74     from collections import Counter
75     # plog(p//q) 这里p直接用频次代替, p*=p*total
76     p_dict=[]
77     temp_li=[]
78     for sample in test_dict:
79         li=[]
80         for key in sample.keys():
81             if key!='word':
82                 li.append(sample[key])
83             temp_li.append(li)
84     trans_li=[[row[i] for row in temp_li] for i in range(len(temp_li[0]))]
85     test_dict=[dict(Counter(row)) for row in trans_li]
86     # print(test_dict)
87     D0=0
88     for test,guess in zip(test_dict,guess_dict):
89         for key in test.keys():
90             D0+=test[key]*math.log(test[key]/guess[key])
91     # print(D0)
92     return D0
93
94
95 def check(test_dict):
96     # 糖丸了, 这可以用两个集合交集大小来确定
97     #correct = len(set(target_words) & set(test_dict))
98     right=0
99     for test in test_dict:
100         if test['word'] in target_words:
101             right+=1
102     return right
103
104 def train(answer_dict,guess_dict,step=100):

```

```

105 test_dict=answer_dict[:target_num]
106 #直接取全集中前n个作为测试集
107
108 rest_dict=answer_dict[target_num:]
109 print(caculate_kl(test_dict,guess_dict))
110
111
112 accuracy = []
113 loss = []
114 for j in range(step):
115     loss.append(caculate_kl(test_dict,guess_dict))
116     kl=[]
117     for i in range(len(test_dict)):
118         new_dict=test_dict[:i]+test_dict[i+1:]
119         kl.append(caculate_kl(new_dict,guess_dict))
120     # print(kl)
121     min_val=min(kl)
122     # print(min_val)
123     index=kl.index(min_val)
124     # print(index)
125     rest_dict.append(test_dict[index])
126     # print(test_dict[index]['word'])
127     test_dict=test_dict[:index]+test_dict[index+1:]
128
129     kl=[]
130     for i in range(len(rest_dict)):
131         new_dict=test_dict+[rest_dict[i]]
132         kl.append(caculate_kl(new_dict,guess_dict))
133     min_val=min(kl)
134     # print(min_val)
135     index=kl.index(min_val)
136     test_dict.append(rest_dict[index])
137     # print(rest_dict[index]['word'])
138     rest_dict=rest_dict[:index]+rest_dict[index+1:]
139     # print('check\nright:')
140     right = check(test_dict)
141
142     print(f"{j}轮训练的正确率为{right}/{target_num}")
143     accuracy.append(right/target_num)
144 return accuracy,loss
145
146
147 def plot_accuracy(accuracy, title="Training Accuracy Over Iterations", xlabel="Iteration
(Step=100)", ylabel="Accuracy"):
148     """
149     绘制迭代准确率曲线图
150
151     参数:
152         accuracy (list): 包含每次迭代准确率的列表
153         title (str): 图表标题
154         xlabel (str): x轴标签
155         ylabel (str): y轴标签
156     """
157
158     iterations = [i for i in range(len(accuracy))]
159

```

```

160     # 设置图表样式
161     plt.figure(figsize=(10, 6))
162     plt.plot(iterations, accuracy,
163             # marker='o',
164             linestyle='--',
165             color='b',
166             label=ylabel)
167
168     # 添加标签和标题
169     plt.title(title, fontsize=14)
170     plt.xlabel(xlabel, fontsize=12)
171     plt.ylabel(ylabel, fontsize=12)
172
173     # 添加网格和优化显示
174     plt.grid(True, linestyle='--', alpha=0.6)
175     plt.legend()
176
177     # 自动调整y轴范围（留10%空白）
178     #plt.ylim(max(0, min(accuracy) - 0.1), min(1, max(accuracy) + 0.1))
179
180     # 显示图表
181     plt.tight_layout()
182     plt.show()
183
184 def guess_game():
185     guess_dict=[]#是一个列表，列表每个元素是一个单词的字典，记录这个单词对答案集的反馈
186     for word in guess_words: #guess是用来放到系统中做测试的单词，不是我们对答案集的估计
187         word_dict={'word':word}
188
189         for target in target_words:#word_dict:word:测试集中随机猜？这里不随机是遍历了；
190             vector=display_vector(word,target)
191             if vector in word_dict.keys():
192                 word_dict[vector]+=1
193             else:
194                 word_dict[vector]=1
195
196         guess_dict.append(word_dict) #猜测集在答案集上的真实分布
197     answer_dict=caculate_answer_dict(guess_dict)#猜测集在估计集上的分布
198     accuracy,loss = train(answer_dict,guess_dict,step=100)
199     plot_accuracy(accuracy)
200     plot_accuracy(loss,title="Training Loss Over Iterations",ylabel="loss")
201
202
203
204 guess_game()

```